# 8

# X.509 Certificates and PKI

In this chapter, we will learn about **X.509 certificates**. Certificates are data structures used for identity presentation and verification. X.509 certificates are crucial for the functioning of TLS and TLS-based protocols, such as HTTPS, where certificates are used to prove the identities of websites. Certificates are also used in secure messaging standards, such as S/MIME; VPN solutions, such as OpenVPN; smart cards, software signing, and so on. X.509 certificates can also optionally be used in IPsec.

We will learn about what certificates consist of, how certificate verification chains are built, and how **Public Key Infrastructure** (**PKI**) works. In the practical part of this chapter, we will learn how to generate certificates and verify certificate chains on the command line and programmatically using C code.

We are going to cover the following topics in this chapter:

- What is an X.509 certificate?
- Understanding certificate signing chains
- How are X.509 certificates issued?
- What are X509v3 extensions?
- Understanding X.509 Public Key Infrastructure
- How to generate a self-signed certificate
- How to generate (issue) a non-self-signed certificate
- How to verify a certificate on the command line
- How to verify a certificate programmatically

## Technical requirements

This chapter will contain commands that you can run on a command line and C source code that you can build and run. For the command-line commands, you will need the `openssl` command-line tool with OpenSSL dynamic libraries. For building the C code, you will need OpenSSL dynamic or static libraries, library headers, a C compiler, and a linker.

We will implement an example program in this chapter, in order to practice what we are learning. The full source code of that program can be found here: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter08`.

## What is an X.509 certificate?

An X.509 certificate is a data structure that is used for identity presentation and identity verification. A certificate may be saved to a file or transmitted via a network, and also as a part of a secure network protocol, such as TLS. An X.509 certificate binds an identity to a public key using a digital signature.

Here is an example text representation of an X.509 certificate:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: ... hex bytes ...
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, O = Let's Encrypt, CN = R3
        Validity
            Not Before: Mar  4 12:43:52 2022 GMT
            Not After : Jun  2 12:43:51 2022 GMT
        Subject: CN = www.openssl.org
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 bit)
                Modulus: ... hex bytes ...
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Server Authentication,
                TLS Web Client Authentication
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Subject Key Identifier:
                ... hex bytes ...
            X509v3 Authority Key Identifier:
```

```
                ... hex bytes ...
            ... more X509v3 extensions ...
```

As we can observe, an X.509 certificate contains the following fields:

- **X.509 version**: Modern X.509 certificates have version 3.
- **Serial number**: The unique number of the certificate among certificates signed by the same issuer.
- **Signature algorithm**: Denotes the cryptographic hash function and digital signature algorithm used to sign the certificate.
- **Issuer**: The entity that signed the current certificate or, according to the X.509 terminology, issued it. The issuer is presented in the **Distinguished Name** (**DN**) format, which will be explained later.
- **Validity**: This is two timestamp fields, `Not Before` and `Not After`, defining when the certificate is valid.
- **Subject**: This is the entity that the certificate identifies. It can be a person, a website, an organization, or something else. The `Subject` field is also represented in the Distinguished Name format.
- **The certificate's public key**: The public key is very important for the identity verification that the certificate provides. Different key types are supported, such as RSA, DSA, ECDSA, and EdDSA. It's important to remember that an X.509 certificate does not contain a private key, only a public key.
- **X509v3 extensions**: They contain additional optional information.
- **The certificate signature**: This is produced by the certificate issuer.

Similar to public and private keys, X.509 certificates are encoded using **Abstract Syntax Notation One** (**ASN.1**) into **Distinguished Encoding Rules** (**DERs**) or **Privacy-Enhanced Mail** (**PEM**) format.

X.509 certificates' `Subject` and `Issuer` fields are represented in the DN format. A Distinguished Name looks like this: `C = US, O = Let's Encrypt, CN = R3`. As we can observe, a DN consists of key-value pairs. The keys have defined meanings, for example, `C` means `Country` and `O` means `Organization`. The most important key is `CN`, which means `Common Name`. It is the main name of the entity identified by the certificate. It can be a website name, such as `www.openssl.org`; a person's name, such as `John Doe`; or, if a certificate is needed for a technical purpose, the name of the certificate itself, for example, `Technical certificate 123` or just `R3`.

Every X.509 certificate has the corresponding private key. That private key is not included in the certificate but matches the public key that is included in the certificate, meaning that a digital signature produced by that private key can be verified by the public key from the certificate. That private key is often called the certificate's private key. The owner of the certificate needs the certificate's private key in order to prove that they really own the certificate and haven't just copied it. Such proof can be established by the owner signing some data using the private key and the verifying party verifying the signature using the public key extracted from the certificate.

An X.509 certificate is usually not secret information and can be freely distributed, similar to a public key. The certificate's private key, on the contrary, is secret information that only the certificate owner should possess, because whoever possesses the private key can claim the identity of the certificate owner and cryptographically prove such an identity. Theft of the private key can lead to identity theft. For example, if someone steals the `www.openssl.org` website certificate's private key and also performs a **DNS poisoning** or **Man in the Middle** (**MITM**) attack, they could set up a false `www.openssl.org` website that will pass the certificate verification. There are ways to mitigate the identity theft described, such as **Certificate Revocation Lists** (**CRLs**) and **Online Certificate Status Protocol** (**OCSP**), but an attacker may still have a time window for their attacks before the compromised certificate is revoked. Here's another example: if a certificate's private key is used for signing email messages, whoever steals the private key will be able to sign emails on behalf of the certificate owner.

When some data is signed by the certificate's private key, it is often said that the data is signed by the certificate. Certificates themselves can also be signed. In fact, each X.509 certificate is signed by some certificate, meaning by that certificate's private key. That signing certificate is specified in the `Issuer` field in the DN format. A certificate can be signed by its own private key. Such a certificate is called a self-signed certificate. Self-signed certificates have the same DN in both the `Subject` and `Issuer` fields. A certificate can also be signed by another certificate, and that other certificate can be signed by a third certificate, and so on. In such a case, we have a certificate signing chain.

## Understanding certificate signing chains

A **certificate signing chain**, also known as a **certificate verification chain**, simply a **certificate chain**, or a **chain of trust**, is an ordered collection of certificates where each certificate is signed by the next certificate in the collection. All except the last certificate, of course. The last certificate is self-signed.

Why are certificate signing chains needed? In order to verify the certificate validity. A curious reader might ask, doesn't the certificate's private key solve this problem? No, it's not so easy. When verifying identity using an X.509 certificate, we have to verify two claims:

1. **That whoever presents the certificate for identification owns the certificate**: This claim is proven using the certificate's private key.
2. **That the presented certificate is valid**: This claim is proven using the certificate signing chain.

It is similar to how you identify yourself with a passport. You can identify yourself only with your own passport, and your passport must be valid.

How does certificate verification work? In order to understand that, we need to learn how certificates are produced or, according to X.509 terminology, *issued*.

Most website certificates on the internet are issued by organizations called **Certificate Authorities** (**CAs**). A CA is an organization that issues certificates and usually charges money for them. Some CAs, for example, Let's Encrypt, issue certificates for free. CAs present themselves as **trusted third parties**. The idea is that you trust CAs to issue certificates and consider certificates issued by those CAs valid. You trust that the CA will check the identities of those who order a certificate from the CA and issue certificates only to those whose identities match the identity written in the certificate's `Subject` field.

Why should you trust some CA organization that you haven't met, have barely heard of, and when you do not know the people working there? CAs give arguments that trust is their business; they cannot issue rogue certificates that do not match the identities of the subjects, otherwise, no one will trust the CAs, and they will go bankrupt. And therefore, you have to trust them. I personally find such argumentation questionable. But you do not have a lot of choice. You (or rather your OS, or your browser) either trust the certificates issued by those CAs, or pretend to trust them, or you cannot access HTTPS sites that use certificates issued by those CAs, which are the majority of the websites on the internet. That's how the modern World Wide Web works.

In order to verify a certificate, you have to build a certificate signing chain finished with a trusted certificate. Let's consider a popular case of website certificate verification. In this case, the first certificate in the chain will be the certificate that identifies the website. That certificate did not sign or issue any other certificates. Certificates that do not sign other certificates are called **leaf certificates**. The website certificate will be signed by the second certificate in the chain. In most cases, that second certificate will not be a self-signed certificate and thus will not finish the certificate chain. Certificates that both sign other certificates and aren't self-signed are called **intermediate CA certificates**. A typical certificate chain starting with a website certificate will have one or two intermediate CA certificates. After all the intermediate CA certificates, the last certificate in the chain will be self-signed. Self-signed certificates that sign other certificates are called **root CA certificates**.

The certificate chain will look like this:

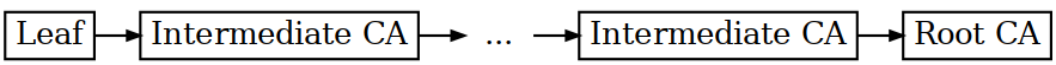


Figure 8.1 – Certificate signing chain, horizontal representation

In various articles on the internet, it is also popular to draw certificate chains vertically, like this:

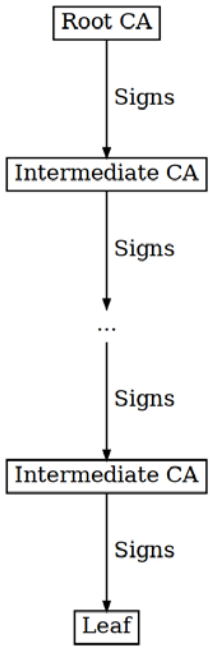


Figure 8.2 – Certificate signing chain, vertical representation

Note that the horizontal representation looks like a linked list, having a certificate to be verified as the first element and a trusted certificate as the last element. That is also how OpenSSL represents certificate signing chains.

In our descriptions and diagrams, we have mentioned intermediate CA certificates. But why are they needed? Why not just sign all leaf (end-entity) certificates with root certificates? There are several reasons for that:

- If an intermediate certificate is compromised, it can be revoked without revocation of the root certificate and other intermediate certificates of the same CA. Hence, the impact of the certificate compromise is limited.
- Many CAs support the automatic issuing of certificates online. This means that the private key of the signing certificate must reside on a server that is directly or indirectly accessible from the internet. Such accessibility is a risk. It is better to put an intermediate certificate at risk than the root certificate. As the root certificate does not sign other certificates very often, its private key can be stored in a safe offline place, for example, on a flash drive in a steel safe, in a locked and guarded room.
- Certificate revocation lists of large CAs can also be large. I have seen CRLs of 50 MB in size. By having several intermediate certificates, a CA can split its pool of issued certificates among intermediate certificates and issue smaller per-intermediate certificate CRLs.

As we have just learned, certificate chains are needed for certificate verification and may contain leaf (end-entity), intermediate, and root certificates. But who verifies those certificate chains and how do verifiers get all the certificates? Anyone who can get the needed certificates and build up a certificate signing chain can verify a certificate. Let's get back to the popular case of website certificate verification. In such a case, it is a browser or another HTTPS client that verifies the certificate. The browser gets the server certificate from the server during the TLS handshake. How does the browser get the root CA certificate? From the browser certificate store or the operating system certificate store. Most operating systems have an OS-wide certificate store that stores certificates and divides them into categories, such as trusted CA certificates, untrusted CA certificates, or client certificates with private keys. Those certificate stores are regularly updated via OS update mechanisms. Some web browsers also have their own certificate stores that can be used in addition to or as a replacement for the OS certificate store. The browser stores are updated by the corresponding browsers, not necessarily when a new browser version is released, but store updates can just be downloaded periodically.

Who decides which certificates are considered trusted? Whoever maintains the certificate store decides how the store is originally populated and which changes to the store (these can be both additions and deletions) are included in the updates. For the OS store, the maintainer is the OS vendor and for the browser store, it is the browser vendor. The OS and browser user can also add and delete certificates from the store. However, not so many users have enough knowledge, need, and desire to change the contents of their certificate stores. Hence, in most cases, the certificate store contains just the certificates that the OS or browser vendor decided to include in it. As a result, the list of trusted CA certificates will consist of certificates that the store vendor chose for you to trust.

The contents of most OS and browser certificate stores are the same or almost the same. The stores are populated by root CA certificates of well-known CAs. If you are visiting a website that has a certificate issued by one of those CAs, your browser will be able to verify the site certificate and will happily show you the web page. If, however, you are visiting an amateur website that uses a certificate not issued by a well-known CA, your browser will not be able to verify the certificate and will show you a warning that can be scary for non-tech-savvy people. In order to avoid the warning, you will have to add the CA certificate that directly, or via intermediate CA certificates, signed the website certificate to the OS or browser certificate store as a trusted certificate. This task can be difficult for a non-tech-savvy person. Therefore, most HTTPS sites on the internet have certificates issued by well-known CAs whose root CA certificates are present in OS or browser certificate stores and are considered trusted. For small amateur sites, it can be too much of a burden to pay money for a site certificate. Fortunately, since 2015, it is possible to get free site certificates from the uncommercial Let's Encrypt CA. Some big commercial CAs sometimes also issue free HTTPS certificates, usually as a trial.

Curious readers may have noticed that certificate stores have a special category for untrusted CA certificates. Why is it needed? The answer is simple. The untrusted CA certificates are intermediate CA certificates that can be used to build a certificate signing chain from a website certificate up to the trusted CA certificate. However, it is quite rare that an OS or browser certificate store contains untrusted CA certificates. How does the browser get the intermediate certificates needed then? Usually, the web server sends the intermediate CA certificates together with the site certificate.

When a certificate is being verified, how is the certificate signing chain built? It is obvious that the first certificate in the chain will be the certificate to be verified, for example, a website certificate. But how do you decide which certificate to put into the chain next? The next certificate can be looked up using two methods. The first method is to use the current certificate's `Issuer` field and look for a certificate with the same DN in the `Subject` field. The `Issuer` field always exists in the certificate, but sometimes several certificates with the same subject can be found. Therefore, searching by the `Issuer` field is ambiguous. The second method is to use the *Authority Key Identifier*" X509v3 extension, which contains the cryptographic hash of the issuer certificate public key. However, X509v3 extensions are optional, hence that extension may be missing from the certificate. Also, sometimes, when a certificate is renewed, it is renewed with the same key but another validity period. Therefore, the second method is also ambiguous. Thus, due to ambiguity, sometimes, in order to build a valid certificate signing chain, it is necessary to analyze several certificate signing paths. If at least one path allows us to build a certificate chain up to a trusted certificate, and all constraints, such as a validity period and constraints imposed by X509v3 extensions, are satisfied, then we can consider the certificate to be successfully verified. OpenSSL supports such an analysis of several signing paths when building a certificate signing chain.

In most cases, the certificate chain finishes with a self-signed root CA certificate. But, strictly speaking, it is not mandatory. OpenSSL supports certificate signing chains that finish with non-self-signed certificates. Using such signing chains, however, is considered a bad practice.

We have now learned a lot about certificate verification. But how are X.509 certificates issued?

## How are X.509 certificates issued?

The X.509 certificate generation procedure consists of several stages:

1. The applicant (future certificate owner) generates the certificate's private and public keys.
2. The applicant generates a **Certificate Signing Request** (**CSR**). The CSR contains the subject, the public key of the future certificate, the X509v3 extensions requested by the applicant, and the CSR signature. The CSR is signed by the certificate's private key.
3. The applicant sends the CSR to a CA for signing.
4. The CA checks the applicant's identity.
5. The CA makes a certificate based on information from the CSR. The CA also adds other information to the certificate, such as the issuer, validity fields, and X509v3 extensions. Finally, the CA signs the certificate.
6. The CA sends the certificate back to the applicant, who then becomes the certificate owner or holder.

Note that no one exposes the private key to another party in the process.

Let's say a few words about how the CA verifies the identity of the applicant in a popular case of issuing a certificate for a website. Some people believe that CAs *always* do a thorough check of the website and its owners, scan the website for malware, visit the website's owners in person, check a lot of their documents, and so on. Usually, it is not so. The identity of most certificates issued to websites is checked automatically and online, without any interaction between the people owning the website and people working in the CA organization. No documents are checked, such as the passports of the owners, contracts for web or DNS hosting, or other documents. What is checked then? It is checked that the applicant controls the website and the DNS of the website domain. The checks are automatic and can be something like this:

- Place a specific file at a specific URL on the website.
- Add a specific DNS record to the website domain.
- Create an email address in the site domain and receive a password at that email.

That's why usual HTTPS certificates are called **Domain-Validated** (**DV**) certificates.

What does such a DV certificate certify when used on a website? That the user's browser is connected to a website with the domain name specified in the certificate, such as `www.openssl.org` – no more, no less. A DV certificate does not assert that a website represents a specific organization, such as the OpenSSL Project. Some people believe in the myth that if a website has a certificate, it is certified and is thus a good site that cannot represent a bad organization, cannot contain malware, cannot attack its users, and all information on that website is true. Unfortunately, it is not so. Sites that are phishing, distributing malware, or doing other criminal activities can also have valid HTTPS certificates, and connections to those sites will be considered "secure" by browsers. It is important to understand that it is only the connection that is secure. The site itself could be very insecure.

Why are DV certificates still useful? Because they protect web browser users from MITM and DNS poisoning attacks. The users can be sure with a high probability that they are really visiting the site that they see in their browser's address line and that it is not a fake site mimicking the original site.

Are there certificates that certify that a website represents a specific organization, not only a specific domain name? Fortunately, yes. Such certificates are called **Extended Validation** (**EV**) certificates. When issuing an EV certificate, the CA does many more checks on the **CSR**. The CA checks that the CSR has come from the organization that owns the domain, the organization name in the CSR `Subject` field matches the real organization name, the organization exists and is properly registered in the government organization register, the organization's address and phone number are real, the person requesting the certificate has authority to do so on behalf of the organization, and so on. EV certificates are only issued to organizations, not to individuals.

EV certificates are significantly more expensive than DV certificates. For example, at the time of writing, one well-known CA sells DV certificates for 8 USD/year and EV certificates for 90 USD/year. Most sites on the internet use DV certificates. EV certificates are used by sites whose identity is especially important, such as the sites of banks.

The EV status of the issued certificate is signaled by an X509v3 extension. Unfortunately, there is currently no standardized X509v3 extension for EV status indication; different CAs use different X509v3 extensions. Therefore, browsers have to implement support for several EV-indicating X509v3 extensions.

Previously, major web browsers indicated that a website had an EV certificate by having a green background in the address bar and showing the site-owning company's name in the address bar. Since 2019, major web browsers stopped using such a strong visual indication. There is no green bar anymore and the company name is only shown if the browser user clicks on the padlock icon in the address bar. Most other software, not web browsers, have never had a strong visual indication of an EV certificate.

There are also **Organization Validation** (**OV**) and **Individual Validation** (**IV**) certificates. They are intermediate solutions between DV and EV certificates. When issuing such certificates, the CA verifies the identity of an organization or an individual and the issued certificate contains the name of the organization or the individual in the certificate's `Subject` field. The difference from EV certificates is that when a CA issues an OV or IV certificate, the CSR does not go through as thorough a validation process as for EV – fewer checks are made. OV and IV certificates are less popular than EV certificates. If an organization wants a higher-rank certificate than DV, it usually chooses an EV certificate rather than OV.

We have already mentioned X509v3 extensions several times. Let's learn a bit more about them.

## What are X509v3 extensions?

X509v3 extensions are additional fields that can be added to an X.509 certificate. X509v3 extensions can impose constraints on certificate usage or provide additional information on the certificate. As an example, let's go through the X509v3 extensions found in the `www.openssl.org` website certificate:

- **Key Usage and Extended Key Usage**: These extensions are constraints that limit certificate usage to certain purposes. It is up to the software that verifies the certificate to enforce X509v3 constraints. If the verifying software does not recognize or enforce certain X509v3 constraints, they will be ignored.
- **The CA:FALSE basic constraint**: This extension asserts that the certificate must not be used to issue other certificates, meaning that the verification software must not consider such a certificate as an intermediate or root CA certificate.
- **Subject Key Identifier and Authority Key Identifier**: Informational extensions that help to look up the issuer certificate from the issued certificate.
- **Authority Information Access**: Informational extensions that help to locate CA resources, such as the OCSP server or CA certificates useful for verification of the current certificate.

- **Subject Alternative Name**: An informational extension that lists alternative domain names of websites that can use this certificate. In addition to `www.openssl.org`, other names could be listed, allowing the certificate owner to use the certificate on sites with other domain names, for example, `mail.openssl.org` or just `openssl.org`.
- **CT Precertificate SCTs**: Information about the certificate entry in the **Certificate Transparency** (**CT**) logs. CT is a monitoring solution that helps to discover the issuing of rogue X.509 certificates.

X509v3 extensions are not mandatory parts of the X.509 certificate format, but their addition makes the usage of certificates more convenient. Also, some software that verifies certificates, such as web browsers, may expect certain X509v3 extensions in the certificate and change the certificate verification result based on the extensions or their absence.

X.509 certificates are useful, but they would not be very useful alone. The certificates are a part of a bigger infrastructure, called Public Key Infrastructure.

## Understanding X.509 Public Key Infrastructure

X.509 **PKI** is a combination of standards, algorithms, data structures, software, hardware, organizations, and procedures needed to create, store, transfer, use, revoke, and otherwise manage X.509 certificates and certificate keys.

Sounds complicated, but we have just learned how X.509 PKI works on the World Wide Web. CAs issue certificates that are used by websites and verified by web browsers. That's how millions of people using the web every day automatically verify the identity of websites. Some websites support the authentication of users using client certificates. In such cases, not only does the website present its certificate but also, the user's web browser presents a client certificate to the site so that the site can verify the client certificate, authenticate, and authorize the user.

X.509 PKI is not only used for the web. X.509 certificates are used for mail transfer, communication of automated computer systems, software signing, and so on.

Some people think that PKI is something mysterious and incomprehensible, but in reality, it's not. It's just managing keys and identities.

We have now learned about X.509 certificates, certificate signing chains, CAs, and PKI. The theoretical part of this chapter is finished. Let's proceed to the practical part and generate some certificates.

## How to generate a self-signed certificate

In order to generate a certificate, we have to generate a key pair, a CSR, and finally, a certificate. The `openssl` tool can generate a self-signed certificate in several ways. One of the ways is to use a single command to generate a key pair and a self-signed certificate. But we will use separate commands because it is a more generic way.

We are going to use the following `openssl` subcommands: `genpkey`, `pkey`, `req`, and `x509`. Their documentation is available on the following man pages:

```
$ man openssl-genpkey
$ man openssl-pkey
$ man openssl-req
$ man openssl-x509
```

This is how we generate a self-signed certificate:

1. First, let's generate a key pair. This time let's use `ED448` as the key type:

   ```
   $ openssl genpkey -algorithm ED448 -out root_keypair.pem
   ```

   We have got no output, meaning that we have got no error.

2. Let's inspect our newly created key:

   ```
   $ openssl pkey -in root_keypair.pem -noout -text
   ED448 Private-Key:
   priv:
      e2:62:21:f0:32:25:20:ca:84:f9:b8:4f:0a:9f:51:
      51:3b:68:d0:0d:3a:91:c9:68:38:b4:2f:d0:53:af:
      62:5a:06:9d:b0:f5:86:11:73:f5:be:39:9a:78:be:
      ec:a2:53:d8:91:ad:8b:e5:2e:e2:b3:a3
   pub:
      70:3c:22:d9:9f:f8:d6:76:e0:4f:46:e8:74:7b:5f:
      98:98:ee:90:49:af:07:ba:05:a4:3b:b3:2c:e3:20:
      1a:00:cf:11:5c:76:93:32:0a:91:14:98:fa:dd:83:
      7b:9c:00:46:c8:d3:df:67:23:ea:e1:80
   ```

   Looks good!

3. Next, let's generate a CSR. To keep things simple, we will not add many X509v3 extensions. We will learn about more sophisticated certificate generation in *Chapter 12*, *Running a Mini-CA*:

   ```
   $ openssl req \
       -new \
       -subj "/CN=Root CA" \
       -addext "basicConstraints=critical,CA:TRUE" \
       -key root_keypair.pem \
       -out root_csr.pem
   ```

   Note that we added the `CA:TRUE` extension. We have to add it for CA certificates.

4. We have got no output and a new file, `root_csr.pem`, containing our CSR. Let's inspect it:

```
$ openssl req -in root_csr.pem -noout -text
Certificate Request:
   Data:
       Version: 1 (0x0)
       Subject: CN = Root CA
       Subject Public Key Info:
           Public Key Algorithm: ED448
               ED448 Public-Key:
               pub:
                   ... hex bytes ...
       Attributes:
           Requested Extensions:
               X509v3 Basic Constraints: critical
                   CA:TRUE
   Signature Algorithm: ED448
   Signature Value:
       ... hex bytes ...
```

5. Now let's generate a self-signed certificate using the CSR and the key pair. Our new certificate will have a validity period of 3,650 days, which is approximately 10 years:

```
$ openssl x509 \
    -req \
    -in root_csr.pem \
    -copy_extensions copyall \
    -key root_keypair.pem \
    -days 3650 \
    -out root_cert.pem
```

Note the `-copy_extensions copyall` switch. It is important to add that switch because the `openssl x509` command, by default, would not copy X509v3 extensions from the CSR to the produced certificate.

6. We have got no output and a new file, `root_cert.pem`, containing our new self-signed certificate. Let's inspect it:

```
$ openssl x509 -in root_cert.pem -noout -text
Certificate:
```

```
    Data:
        Version: 3 (0x2)
        Serial Number:
            ... hex bytes ...
        Signature Algorithm: ED448
        Issuer: CN = Root CA
        Validity
            Not Before: Mar 27 22:25:17 2022 GMT
            Not After : Mar 24 22:25:17 2032 GMT
        Subject: CN = Root CA
        Subject Public Key Info:
            Public Key Algorithm: ED448
                ED448 Public-Key:
                pub:
                    ... hex bytes ...
        X509v3 extensions:
            X509v3 Basic Constraints: critical
                CA:TRUE
            X509v3 Subject Key Identifier:
                ... hex bytes ...
    Signature Algorithm: ED448
    Signature Value:
        ... hex bytes ...
```

Note that the `Issuer` and `Subject` fields are the same in the certificate. It is an indication of a self-signed certificate.

We have now successfully generated a self-signed certificate, which we will use as a root CA certificate. Let's now generate a couple of non-self-signed certificates.

## How to generate a non-self-signed certificate

When generating a self-signed certificate, we used a generic approach instead of long combined commands. Therefore, the generation of non-self-signed certificates will be very similar. We will now generate a couple of non-self-signed certificates. We will use one as an intermediate CA certificate and another as an end-entity leaf certificate.

Let's proceed with the certificate generation:

1. First, let's generate a key pair for the intermediate CA certificate:

```
$ openssl genpkey \
    -algorithm ED448 \
    -out intermediate_keypair.pem
```

2. Next, generate a CSR:

```
$ openssl req \
    -new \
    -subj "/CN=Intermediate CA" \
    -addext "basicConstraints=critical,CA:TRUE" \
    -key intermediate_keypair.pem \
    -out intermediate_csr.pem
```

   Note that we used a different `Subject` for the intermediate CA certificate.

3. Now we will issue the intermediate CA certificate, signing it with the private key of the root certificate:

```
$ openssl x509 \
    -req \
    -in intermediate_csr.pem \
    -copy_extensions copyall \
    -CA root_cert.pem \
    -CAkey root_keypair.pem \
    -days 3650 \
    -out intermediate_cert.pem
```

   Note that we used `-CA` and `-CAkey` switches instead of the `-key` switch. The `-CA` switch is needed for copying the CA certificate's `Subject` value into the issued certificate's `Issuer` field, and also for taking other necessary information from the issuing certificate, for example, the *Authority Key Identifier*, if the corresponding X509v3 extension is used.

4. Let's inspect the issued intermediate CA certificate:

```
$ openssl x509 -in intermediate_cert.pem -noout –text
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number:
```

```
            ... hex bytes …
        Signature Algorithm: ED448
        Issuer: CN = Root CA
        Validity
            Not Before: Mar 27 23:11:35 2022 GMT
            Not After : Mar 24 23:11:35 2032 GMT
        Subject: CN = Intermediate CA
        Subject Public Key Info:
            Public Key Algorithm: ED448
                ED448 Public-Key:
                pub:
                    ... hex bytes ...
        X509v3 extensions:
            X509v3 Basic Constraints: critical
                CA:TRUE
            X509v3 Subject Key Identifier:
                ... hex bytes ...
            X509v3 Authority Key Identifier:
                ... hex bytes ...
    Signature Algorithm: ED448
    Signature Value:
        ... hex bytes ...
```

Note that this certificate has different `Issuer` and `Subject` fields, indicating that it is not a self-signed certificate.

5. Now let's issue a leaf certificate and sign it with the private key of the intermediate CA certificate. We will do it very similarly to issuing the intermediate CA certificate:

```
$ openssl genpkey -algorithm ED448 -out leaf_keypair.pem
$ openssl req \
    -new \
    -subj "/CN=Leaf" \
    -addext "basicConstraints=critical,CA:FALSE" \
    -key leaf_keypair.pem \
    -out leaf_csr.pem
$ openssl x509 \
    -req \
    -in leaf_csr.pem \
    -copy_extensions copyall \
```

```
    -CA intermediate_cert.pem \
    -CAkey intermediate_keypair.pem \
    -days 3650 \
    -out leaf_cert.pem
```

Note that this time we set `CA:FALSE` instead of `CA:TRUE` because leaf certificates are not supposed to issue other certificates.

6. Let's inspect the produced leaf certificate:

```
$ openssl x509 -in leaf_cert.pem -noout –text
Certificate:
   Data:
       Version: 1 (0x0)
       Serial Number:
           ... hex bytes ...
       Signature Algorithm: ED448
       Issuer: CN = Intermediate CA
       Validity
           Not Before: Mar 27 23:34:19 2022 GMT
           Not After : Mar 24 23:34:19 2032 GMT
       Subject: CN = Leaf
       Subject Public Key Info:
           Public Key Algorithm: ED448
               ED448 Public-Key:
               pub:
                   ... hex bytes ...
        X509v3 extensions:
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Subject Key Identifier:
                ... hex bytes ...
            X509v3 Authority Key Identifier:
                ... hex bytes ...
   Signature Algorithm: ED448
   Signature Value:
       ... hex bytes ...
```

Looks good, the leaf certificate is issued by the intermediate CA certificate.

Now let's verify the leaf certificate using the other two certificates.

## How to verify a certificate on the command line

Certificate verification on the command line can be done using the `openssl verify` command. Its documentation can be found on the `openssl-verify` man page.

Let's verify the leaf certificate that we have just generated. We will consider our root CA certificate a trusted certificate. Our intermediate CA certificate will be considered untrusted, but it will help us to build the certificate signing chain.

Here is how we can verify the leaf certificate on the command line:

```
$ openssl verify \
    -verbose \
    -show_chain \
    -trusted root_cert.pem \
    -untrusted intermediate_cert.pem \
    leaf_cert.pem
leaf_cert.pem: OK
Chain:
depth=0: CN = Leaf (untrusted)
depth=1: CN = Intermediate CA (untrusted)
depth=2: CN = Root CA
```

Note the `-trusted` and `-untrusted` switches. The `-trusted` switch is used to specify a file containing one or more trusted certificates. In order to successfully verify a certificate, `openssl verify` has to build a certificate signing chain from the certificate being verified up to a trusted certificate. The `-untrusted` switch is used to specify a file with one or more untrusted certificates. Untrusted certificates are useful as intermediate certificates in the certificate chain. Both `-trusted` and `-untrusted` switches can be used several times to specify several files.

`openssl verify` has reported `OK`, meaning that the certificate verification has succeeded. `openssl verify` has also printed the certificate chain that it has built in order to verify the certificate.

We have successfully verified a certificate on the command line. Let us now learn how to verify a certificate programmatically.

## How to verify a certificate programmatically

OpenSSL 3.0 provides only one API for certificate verification. The API consists of functions starting with the `X509_` prefix.

We are going to develop a small program that verifies the leaf certificate that we have just generated, similar to how `openssl verify` does so.

Here are some relevant manual pages for the API that we are going to use:

```
$ man X509_STORE_new
$ man X509_STORE_load_file
$ man DEFINE_STACK_OF
$ man PEM_read_x509
$ man X509_STORE_CTX_new
$ man X509_verify_cert
$ man X509_STORE_CTX_get_error
$ man X509_free
```

Our program will take three command-line arguments:

1. The name of the file containing trusted certificates
2. The name of the file containing untrusted certificates
3. The name of the file containing the target certificate that is going to be verified

Our high-level implementation plan will be as follows:

1. Load trusted certificates.
2. Load untrusted certificates.
3. Load the target certificate.
4. Create and initialize the certificate verification context.
5. Verify the certificate.

Now it's the time to implement the plan.

## Implementing the x509-verify program

Let's develop our `x509-verify` program according to the plan:

1. First, we have to load trusted certificates. We will need to supply a collection of trusted certificates to the verification function as an `X509_STORE` container. Fortunately, OpenSSL provides a convenient function, `X509_STORE_load_file()`, for loading certificates from a PEM file to `X509_STORE`:

```
const char* trusted_cert_fname = argv[1];
X509_STORE* trusted_store = X509_STORE_new();
X509_STORE_load_file(trusted_store, trusted_cert_fname);
```

2. Next, we must load untrusted certificates. We will need to supply untrusted certificates to the verification function as a `STACK_OF(X509)` object. `X509` is an OpenSSL type that represents an X.509 certificate. `STACK_OF` is a macro that constructs an OpenSSL stack type. The `STACK_OF` macro is similar to the `std::stack` template type in C++. The macro takes the stack element type as a parameter. OpenSSL uses `STACK_OF` stacks a lot in its API and internal code. `STACK_OF(X509)` objects are often used as certificate signing chains or just as certificate lists. OpenSSL functions that work with stacks, such as `sk_X509_new_null()` and `sk_X509_push()`, are also macros. They have the `sk_` prefix and their names are parameterized with the stack element type name. For example, `sk_` functions working with `STACK_OF(X509)` have the `X509` substring in their names. In other words, the names of those functions start not just with `sk_` but with `sk_X509_`. As the `X509` substring is, in this case, just a parameter in a stack function name, you will not find a man page called `sk_X509_push`. But you will find the `sk_TYPE_push` man page, which is an alias for the `DEFINE_STACK_OF` man page.

3. Unfortunately, there is no convenient function in OpenSSL that would load certificates to `STACK_OF(X509)` in one operation. Therefore, we must load certificates one by one using the `PEM_read_X509()` function. First, let's find out the length of the file containing the untrusted certificates:

```
const char* untrusted_cert_fname = argv[2];
FILE* untrusted_cert_file =
    fopen(untrusted_cert_fname, "rb");
fseek(untrusted_cert_file, 0, SEEK_END);
long untrusted_cert_file_len = ftell(untrusted_cert_
file);
fseek(untrusted_cert_file, 0, SEEK_SET);
```

4. Then, let's load certificates one by one into a `STACK_OF(X509)` structure:

```
STACK_OF(X509)* untrusted_stack = sk_X509_new_null();
while (ftell(untrusted_cert_file) < untrusted_cert_file_
len){
    X509* untrusted_cert =
        PEM_read_X509(untrusted_cert_file, NULL, NULL,
NULL);
    sk_X509_push(untrusted_stack, untrusted_cert);
}
```

We cannot just use the `while(!feof(untrusted_cert_file))` loop condition because the `feof()` function only detects the end of the file when there is an attempt to read beyond the end of the file. And if we call the `PEM_read_X509()` function at the end of the file before the end of the file condition is detected, the `PEM_read_X509()` function will fail. Therefore, we rely on the file position in the loop.

5. After the trusted and untrusted certificates are loaded, it is time to load the target certificate that has to be verified. It is just one certificate; thus, we can conveniently load it using one `PEM_read_X509()` function call:

```
const char* target_cert_fname = argv[3];
FILE* target_cert_file = fopen(target_cert_fname, "rb");
X509* target_cert =
    PEM_read_X509(target_cert_file, NULL, NULL, NULL);
```

6. All the certificates are loaded. Now it's time to create and initialize the certificate verification context, which is an object of the `X509_STORE_CTX` type:

```
X509_STORE_CTX* ctx = X509_STORE_CTX_new();
X509_STORE_CTX_init(
    ctx, trusted_store, target_cert, untrusted_stack);
```

7. The decisive moment – certificate verification using the verification context:

```
int err = X509_verify_cert(ctx);
```

The function returns `1` if the certificate has been verified successfully. Other return values indicate failure.

8. If a failure happened, the exact error can be obtained using the `X509_STORE_CTX_get_error()` and `X509_verify_cert_error_string()` functions:

```
if (err != 1) {
    int error_code = X509_STORE_CTX_get_error(ctx);
    const char* error_string =
        X509_verify_cert_error_string(error_code);
    fprintf(
        stderr,
        "X509 verification error: %s\n",
        error_string);
}
```

9. After the finished verification, the objects that were in use must be freed:

```
X509_STORE_CTX_free(ctx);
X509_free(target_cert);
sk_X509_pop_free(untrusted_stack, X509_free);
X509_STORE_free(trusted_store);
```

The complete source code of our `x509-verify` program can be found on GitHub in the `x509-verify.c` file: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter08/x509-verify.c`

## Running the x509-verify program

Let's run our x509-verify program and verify the leaf certificate, using the generated root CA certificate as a trusted certificate and the intermediate CA certificate as an untrusted certificate:

```
$ ./x509-verify \
    root_cert.pem \
    intermediate_cert.pem \
    leaf_cert.pem
Verification succeeded
```

As we can observe, certificate verification has succeeded, as we expected.

Let's also check a failing case. This time, we will not provide the intermediate CA certificate to our program so it won't be able to build a certificate signing chain up to the trusted certificate. We still have to provide three arguments to the program; thus, we will just provide the `leaf_cert.pem` argument two times:

```
$ ./x509-verify \
    root_cert.pem \
    leaf_cert.pem \
    leaf_cert.pem
X509 verification error: unable to get local issuer certificate
Verification failed
```

We observe verification failure in this case, just as expected.

As we can see, our `x509-verify` program supports both successful and failing cases.

This section concludes the practical part of the chapter. Let's proceed to the summary.

## Summary

In this chapter, we learned about the concept of X.509 certificates, why are they needed, and what kind of information they contain. We also learned about certificate signing chains and their role in certificate verification. Then we learned about CAs, as well as the differences between root and intermediate CA certificates. We also learned about the process of issuing X.509 certificates and several types of certificates, such as domain validation and EV types. Then we learned about X509v3 extensions. We finished the theoretical part of the chapter by learning about the concept of PKI.

In the practical part of the chapter, we learned how to generate self-signed and non-self-signed certificates. Then, we learned how to verify certificates, both on the command line and programmatically using C code.

In the next chapter, we will learn about setting up TLS connections and sending data over them.